**Multicore Programming Project 1**

담당 교수 : 박성용

이름 : 성진규

학번 : 20191598

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* **(MyShell을 만드는 전체적인 개요에 대해서 작성하면 됨.)**

이번 프로젝트에서는 Linux 환경의 bash와 거의 유사하게 동작하는 myshell을 C언어를 이용해 구현한다.

Phase1에서는 built-in command의 동작 중 cd, history, exit를 직접 구현함과 동시에 기본적인 fork() 호출을 통한 child process 생성 및 child process에서의 program loading & running을 이용해 mkdir, rmdir, touch, cat, echo, ls 같은 command도 적절히 실행되게 구현해야 한다.

Phase2에서는 Phase1에서 구현한 command들을 pipeline으로 실행할 수 있도록 구현하는 것이 목적이다. 이 때, file descripter 개념에 유의한다.

Phase3에서는 Background 실행되는 프로세스들에 대한 처리를 구현한다. command 끝에 &를 붙이면 해당 프로세스는 Background에서 실행된다. 이와 관련하여 jobs, bg, fg, kill command도 함께 구현한다. signal와 handler에 대한 이해, signal blocking에 유의하여 구현해야한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

cd **:** myshell> cd directory\_path 로 입력하였을 때, 현재 경로를 사용자가 입력한 경로로 바꾼다. myshell> cd 로 입력하였을 때, 홈 디렉토리로 이동한다.

history : myshell> history 로 입력하였을 때, myshell에서 여태까지 입력되었던 command들을 전부 출력한다. 이 경우 연속된 두 command가 다른 경우에만 history에 추가한다. command들은 한 번 저장되면 myshell 종료 후 다시 실행한 후에도 그 이력이 유지되어야 한다.

!! : history에서 가장 최근 command로 치환된다. 예를 들어, 가장 최근 command가 ls라고 가정하자. 그러면 myshell> !!abc 로 입력하면 command는 lsabc로 치환되어 입력된다.

!# : history에서 #번째 command로 치환된다. 예를 들어, 4번째 command가 ls라고 가정하자. 그러면 myshell> !4abc 로 입력하면 command는 lsabc로 치환되어 입력된다.

exit : myshell> exit 로 입력하면 myshell 프로세스가 종료된다.

ls, mkdir, rmdir, touch, cat, echo : fork() 후에 자식 프로세스에서 execvp() 를 이용하여 실행한다. 따라서 별다른 구현없이도 bash에서와 동일하게 동작한다.

1. Phase 2

pipeline을 구현한다. 예를 들어, myshell> cat filename | grep “ab c” | sort 로 cmdline을 입력한다고 생각해보자. 그러면, 어떤 파일의 내용에서 ab c를 포함하는 라인을 정렬하여 stdout에 출력해주도록 구현해야한다.

1. Phase 3

적절한 모든 실행은 job으로 관리된다. job은 running, stopped, terminated 세 가지 상태를 가진다. terminated된 job은 리스트에서 제거된다.

jobs : 전체 job의 목록을 확인 할 수 있다.

fg %# : background job을 running foreground job으로 바꾼다.

bg %# : foreground job을 running background job으로 바꾼다.

kill %# : 사용하면 실행 중인 background job을 종료시킬 수 있다.

foreground job은 ctrl + c, ctrl + z 로 보내지는 SIGINT, SIGTSTP signal에 적절히 반응하되, myshell이 이 signal 에 반응하지 않아야 한다.

모든 phase에 걸쳐서, 부적절한 입력에 들어오면 예외 메세지를 출력해야하며, 에러에 의해서 프로그램이 종료되어서는 안된다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술**
* **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

입력된 cmdline이 parseline() 에 의해서 적절히 argv[], argc를 설정하게 된다. argv[0]에는 우리가 실행하고 싶은 명령어가 들어있다. argv[0]이 built-in command인 경우, 즉 history, exit, cd 인 경우는 fork()하지 않는다.

built-in command가 아닌 경우는 fork() 함수를 호출하여 child process를 생성한다. 이후 생성된 child process에서 execvp() 함수를 호출하여 process를 특정 command를 동작하도록 child process를 overwrite한다.

* + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow

child process에서 execvp()을 이용해 모든 작업이 실행된 이후에 child process는 종료되며, parent process는 SIGCHLD signal을 전달받게 된다.

이 상황에서 종료된 child process를 reaping 해주기 위해서 parent process에서 wait(&status) 함수를 호출하면 된다.

Kernel의 동작에 의해 parent와 child process의 실행 우선순위가 미결정된 상태이므로 각 경우에 대해서 flow를 정리해보면 아래와 같다.

Case1. Call fork() → child process 생성 → execvp() → child process 종료 → child process가 zombie가 됨 → parent process 실행 (Call wait()) → zombie process reaping

Case2. Call fork() → child process 생성 → execvp() → parent process 실행 (Call wait()) → child process 종료 → child process reaping

즉, parent process가 child process 종료 이전에 실행되든 이후에 실행되든 wait() 함수에 의해서 문제없이 signal handling(catching)이 이뤄지는 것을 알 수 있다.

* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)

Step1. ‘|’ 를 기준으로 buf를 분리하여 splited\_bufs[MAXBUF]에 저장한다. 이 때, pipeline을 구성하는 command들의 개수를 max\_idx를 이용해 관리한다.

Step2. max\_idx를 사용하여 for 반복문으로 각 command에 접근한다.

Step3. 접근한 각 command 들에 대해서 parseline() 함수를 이용해 argv[]와 argc를 생성한다.

Step4. 생성한 argv, argc를 고려하며, 해당 command가 적절한 pipeline을 구성할 수 있는 경우는 fork() 로 child process를 생성하여 그 곳에서 실행한다. 이 때, file descripter와 pipe(), dup2() 함수들을 사용하여 생성된 child process들의 출력과 입력을 연결하여 pipeline을 구축한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

pipeline 개수가 0개인 경우 : splited\_bufs[0]에만 값이 저장되게 구현하였다. 따라서, 기존 phase1과 동일하게 정상적으로 동작했다.

pipeline 개수가 (n > 0) 개이면서 적절한 pipeline 입력으로 들어오는 경우 : splited\_bufs[0 ~ n-1] 까지 command 값들이 저장되어 있다. 각 입력을 반복하면서 child process를 생성하고, 생성후 실행하기 전에 file descripter를 이전 command에 대한 child process의 출력을 현재 command에 대한 child process의 입력과 연결하고, 반대로 현재 command에 대한 child process의 출력을 다음 child process의 입력으로 연결해준다. 이를 반복하면 전체 pipeline이 구축된다. parent process에서는 사용하지 않는 file descripter를 close 해준다.

pipeline 개수가 (n > 0) 개이면서 적절하지 않은 pipeline 입력이 들어오는 경우 : buf를 모든 입력에 대해서 ‘|’ 기준으로 분리한 다음 그 사이의 command를 저장하기 때문에 별다른 예외처리 없이도 적절하지 않은 command에 대해서 phase1과 동일하게 적절하지 않은 command라는 점을 출력하여주고, 전체 child process들은 parent process에 의해서 전부 Reaping 된다.

* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명

&가 command에 유효하게 입력된 경우, 해당 command는 background에서 실행되어야 한다. 따라서, 우선 주어진 &를 포함한 command가 유효한지를 우선 판단한다.

command가 유효하지 않은 경우, Phase1, 2에서의 구현에 의해서 command가 실행되지 않으므로 별다른 처리를 해줄 필요가 없다.

command가 유효한 경우, foreground 실행과 다르게 parent process에서 해당 background process의 실행 종료를 기다리지말고 바로 다음 입력을 받을 수 있도록 해야한다.

background 실행이 종료되면 parent process는 이를 reaping 해주고 Jobs에서 지운다. 만일 실행이 종료되지 않는 프로세스라면 kill %(#) built-in command를 이용해서 이를 죽일 수 있어야 한다. 즉, Reaping 시점이 프로그램 실행 전반에 걸쳐 발생할 수 있기 때문에, SIGCHLD handler를 install하여서 구현한다.

* 1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명
    1. int builtin\_command(char \*\*argv);

argv[0]이 built-in command (exit, history, cd)인 경우 해당하는 동작을 수행하고 1을 리턴한다. 그렇지 않은 경우 0을 리턴한다.

* + 1. void eval(char \*cmdline);

cmdline을 parseline()을 사용해서 argv에 적절히 저장하였다는 사실을 생각해보자. builtin\_command() 를 호출하였을 때, 0이 리턴되는 경우에는 /bash/bin 에서 해당 command를 실행해야하므로, 우선 fork()를 호출하여 child process를 만들어준다. child process에서 execvp(argv[0], argv)를 호출하여 해당 command를 실행한다.

* + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow

child process에서 특정 command가 execvp()를 통해서 완전히 실행되고 해당 프로세스가 종료되고 나면 SIGCHLD signal이 parent process로 전달된다. 이때, unistd.h 헤더파일에 존재하는 wait()를 호출하여서 단순히 reaping 해주면 된다. 이 때, wait()의 리턴값이 0보다 작으면 (정상적인 reaping이 실패한 경우) 에러 메시지를 출력하도록 unix\_error()를 호출하게 구현한다.

Foreground process는 완전히 종료된 이후에 다음 입력을 받기 때문에 전체 myshell 동작 과정에서 Reaping 시점이 정해져있으므로 Handler install은 불필요하다.

* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)

이 부분의 구현 사항은 별도의 함수 구현없이 eval() 함수 내부 구조를 적절히 수정하여서 구현하였다. 각 Step 별로 구현사항을 정리하였다.

Step1. ‘|’ 를 기준으로 buf를 분리하여 splited\_bufs[MAXBUF]에 저장한다

string.h 에 선언되어있는 char\* strtok(char\* str, char\* delimiters) 을 사용한다. strtok(buf)로 처음 호출하면, “|”가 있는 위치를 왼쪽부터 찾고, 찾은 경우 문자열 “|”를 \0으로 바꾼다. 이 후 strtok(NULL, “|”)를 호출하면 이전에 찾은 구분자 뒤에서 부터 다시 찾게된다. 이를 반복하는 코드를 eval() 실행하여 cmdline을 command 별로 분리하여 splited\_bufs[MAXBUF]에 저장해둔다. 이 때 max\_idx로 저장하며 최대 인덱스를 저장해둔다.

Step2 & 3. max\_idx를 사용하여 for 반복문으로 각 command에 접근하고, 접근한 각 command 들에 대해서 parseline() 함수를 이용해 argv[]와 argc를 생성한다.

분리한 command들에 for (int i = 0; i < max\_idx; i++)로 접근한다. 따라서 phase1에서 parseline을 호출하고 builtin\_command()를 실행하는 부분 전체를 반복문으로 감싸서 해당 작업을 각 command에 대해서 전부 실행하게 구현한다.

Step4. 생성한 argv, argc를 고려하며, 해당 command가 적절한 pipeline을 구성할 수 있는 경우는 fork() 로 child process를 생성하여 그 곳에서 실행한다. 이 때, file descripter와 pipe(), dup2() 함수들을 사용하여 process 사이의 입력과 출력을 연결해주며 pipeline을 구축한다.

구현 file descirpter를 eval() 호출시에 곧바로 생성한다. int fd[2] 가 그것이다. 이 때 fd[1] 을 출력용, fd[0]을 입력용으로 하였고, WRITE = 1, READ = 0으로 define하여 코드의 가독성을 높였다. 또한 이전 프로세스의 file\_descripter를 이후의 프로세스에 넘겨주기 위한 int fd\_backup을 선언하였다.

우선 dup2(fd\_backup, STDIN\_FILENO) 를 하여서 이전 프로세스의 출력 fd를 현재 프로세스의 입력으로 연결해준다. 만일 현재 command가 pipeline의 마지막 command가 아닌 경우는 dup2(fd[WRITE], STDOUT\_FILENO) 를 호출하여 출력을 다음 프로세스로 연결해준다. 마지막 command인 경우는 이 작업을 하지 않는다. 이후 사용하지 않는 fd[READ]를 close해준다. close(fd[READ])가 그것이다.

이러한 파이프 연결 이후 execvp()를 사용하여 command를 실행해준다.

parent process에서는 reaping 하기 전에 더 이상 사용하지 않는 fd[WRITE]를 닫아준다. 즉, close(fd[WRITE])를 호출한다. 마지막으로 fd\_backup = fd[READ] 를 호출여서 다음 프로세스에 넘겨주기 위한 밑작업을 해둔다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

앞서 설명했듯, 반복문을 통해서 접근하면서 command를 저장해둔 배열의 개수에 상관없이 동작하게 구현하였으므로 개수에 따라서 별도로 handling 해주지 않아도 모든 경우에 적절히 동작한다.

* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명

우선 job\_t 구조체를 하나 선언한 뒤, 구조체 배열인 Jobs를 이용해서 모든 job들을 관리한다. 구조체 멤버로는 int order, int job\_id, char state, char argv[]가 있다.

order : 전체 jobs 사이에서 지정된 해당 job의 번호를 저장

job\_id : 해당 job의 pid를 저장

state : f (foreground), r(running), s(stopped), t(terminated) 상태 표시

argv : job에 대응하는 입력 command line 저장

전체 argv는 job 목록을 출력하는 \_job() 함수를 호출하였을 때, 혹은 job을 background로 만들어주는 \_bg() 함수와 foreground로 만들어주는 \_fg() 호출이 일어날 때, job들에 대응하는 입력 커맨드를 출력해주기 위해 필요하다. main의 char \*argv[]에 있는 모든 문자열들을 strcat() 함수를 사용하여 하나로 묶어주는 작업을 add\_job(pid, state, argv) 함수에서 한 뒤 구조체 배열에 저장한다. add\_job()은 구조체 배열에 job을 추가해주는 함수이다.

이제 Bakcground의 동작을 구현한 방법에 대해서 알아보자.

우선 &로 들어오는 입력 커맨드들은 parseline() 함수에 의해서 bg 변수가 1로 설정된다. bg 변수가 1이 되는 경우 parent process에서 wait하지 않고, 그냥 다음 입력을 받도록 구현했다.

이 때, cmdline이 ./a.out& 같이 &와 command 사이의 공백이 없이 들어오는 경우는 phase2까지는 유효했던 parseline() 함수가 적절히 처리하지 못했다. &는 항상 argv[] 배열의 마지막 요소인 것을 이용해서 parseline의 마지막 부분에 이를 처리해주는 코드를 작성하였다. 간단히 요약하면, argv[]의 마지막 요소의 마지막 글자 (‘\0’ 직전) 가 항상 &일 것이므로, &를 하나 지우고 bg 변수를 1로 설정해준다.

parent process에서 add\_job(pid, state, argv) 를 호출하여 background process를 구조체 배열 Jobs에 추가해준다. Background process는 Reaping 시점이 myshell 동작 과정 전반에 걸쳐서 일어날 수 있기 때문에 waitpid() 함수를 parent process에서 호출하지 말고, parent process에서 SIGCHLD 신호를 받을 때 까지 기다리고, 신호를 받으면 Handler에서 waitpid()를 이용해 Reaping하는 방식으로 구현하였다.

background process가 절대로 종료되지 않는 경우, kill을 이용해서 죽여야하므로 이를 구현한다. void \_kill(int target) 함수는 target(>0) order를 가지는 job에 Kill() 함수를 호출하여 SIGKILL 신호를 보낸다. SIGKILL 신호를 받으면 process가 종료되며 이에 따라서 SIGCHLD가 보내져서 SIGCHLD handler에서 종료된 process를 reaping한다.

Jobs에서도 Reaping된 process를 제거해야하며, 이 동작도 SIGCHLD handler에서 처리해준다.

1. **구현 결과**
   1. **Flow Chart**

* **2.B.개발 내용에 대한 Flow Chart를 작성.**
* **(각각의 방법들에서 추가된 내용(fork, pipeline, background)만 특성이 잘 드러나게 그리면 됨.)**

1. 도표이(가) 표시된 사진

   자동 생성된 설명**Phase 1 (fork)**
2. 도표이(가) 표시된 사진

   자동 생성된 설명**Phase 2 (pipeline)**
3. **도표이(가) 표시된 사진

   자동 생성된 설명Phase 3 (background)**